Un système de vérification de processus parallèles et communicants

Valérie Lecompte, Eric Madelaine, Didier Vergamini

INRIA

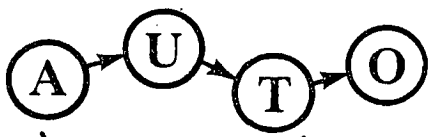
UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports Techniques

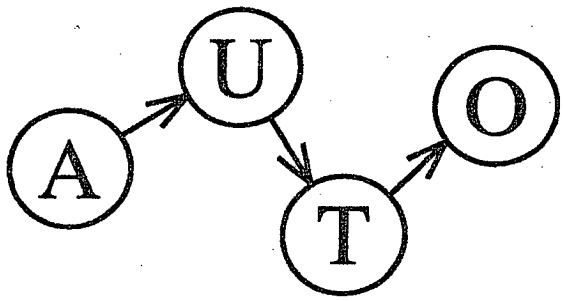
N° 83



UN SYSTÈME DE VÉRIFICATION DE PROCESSUS PARALLÈLES ET COMMUNICANTS

Valérie LECOMPTE
Eric MADELAINE
Didier VERGAMINI

Mars 1987



Un système de vérification de processus parallèles et communicants

A verification system for parallel and communicating processes

Valérie Lecompte *
Eric Madelaine **
Didier Vergamini *

*CMA-ENSMP
Place Sophie Laffitte
Sophia Antipolis
06565 Valbonne Cedex (France)

**INRIA
Route des Lucioles
Sophia Antipolis
06565 Valbonne Cedex (France)

Résumé

Ce rapport décrit les principes d'utilisation de AUTO, un système de vérification pour un calcul de processus parallèles et communicants proche de MEIJE. Nous avons implanté des algorithmes qui permettent de construire et de visualiser des automates finis afin de pouvoir utiliser et expérimenter des principes de vérification tels que le quotient d'un automate par un critère d'observation.

Abstract

This report describes the utilisation principles of AUTO, a system of verification for a parallel and communicating processes calculus à la MEIJE. We have implemented algorithms allowing to build and visualize finite state automata in order to use and experiment verification principles such as the quotient of an automaton by an observational criterion.

1 Introduction.

L'étude des calculs de processus parallèles et communicants est fondée sur l'interprétation des termes d'une algèbre comme des systèmes de transitions. Ce type de sémantique est décrite dans le cours de G. Plotkin [Pl] et a été utilisé pour construire SCCS [Mi] et MEIJE [Au&Bo]. AUTO est un système permettant de manipuler des termes d'une algèbre proche de MEIJE qui dénotent des systèmes de transitions finis. On a ainsi implanté divers outils de vérification tels qu'ils ont été définis dans [Bo,Ve]. Le lecteur est renvoyé à ces derniers articles pour ce qui concerne la théorie de la vérification que le système AUTO met en œuvre.

2 Lancement du système.

AUTO est un système écrit en LELISP, son implantation a été développé sous UNIX sur une SM90. Son portage sur SPS9, SUN3 et VAX8650, tous trois sous UNIX, n'a posé aucun problème, le système fonctionne de manière identique sur toutes ces machines. Aussi toutes les références au système d'exploitation seront des références à UNIX.

La commande **auto** permet de lancer AUTO. En fait, elle lance LELISP qui recharge la core-image spécifique de AUTO. Si on désire préciser la taille de la zone des listes que l'on veut, il suffit de la donner en argument à **auto**.

```
sps9$ auto 40

                        AUTO
                Version 1.6 (Janvier 87)
0
```

3 Le toplevel du système.

Le système AUTO permet de construire des termes d'un calcul de processus, et de calculer les automates associés à ces termes. L'utilisateur dispose d'un interprète de commandes qui lui permet de gérer un environnement de liaisons "variable-objet associé". AUTO connait un langage que nous appelons "toplevel", qui permet d'évaluer d'une part des commandes, d'autre part des expressions construites avec des fonctions et des objets de base. Certaines commandes permettent d'utiliser des analyseurs d'autres langages tels que celui de description de réseaux d'automates ou celui des critères d'abstraction.

Les objets sont typés, les différents types possibles étant:

- le type booléen,
- le type entier,
- le type chaîne de caractères,
- le type liste d'objets,
- le type action,
- le type MEIJE0,
- le type nfauto,
- le type automate,
- le type critère.

De plus, les liaisons réalisées au toplevel sont statiques, c'est-à-dire que les références aux variables libres d'un terme sont considérées comme des références aux variables liées dans l'environnement du toplevel lorsque l'on écrit le terme.

Les exemples donnés dans ce rapport seront le plus souvent empruntés au problème des philosophes mangeurs de spaghetti.

3.1 Eléments lexicaux, rattrapage des erreurs de syntaxe.

Un texte du langage de commande comprend des identificateurs, des nombres, des chaînes de caractères, des symboles, séparés par des séparateurs. Les nombres et les chaînes de caractères limitées par le caractère "" sont les unités lexicales classiques. Les caractères composant un identificateur peuvent être tout caractère alphanumérique , le caractère "-" ou bien le caractère "_", le premier étant obligatoirement une lettre. Les symboles sont des suites de caractères tels que "[]<>/\=.+:^!?", ils servent le plus souvent à écrire des opérateurs infixes. Les séparateurs sont les caractères `espace`, `retour-chariot` et `tabulation` et les commentaires: un commentaire commence par le caractère "%" et se termine par un retour-chariot.

Toute commande est une suite d'élements lexicaux se terminant par ";" ou bien ";;". Lorsqu'une erreur de syntaxe se produit, l'interprète attend la suite de caractères ";;" pour se rattraper.

```
@ 3 "foo" ;
** cy-top1 : syntax error ;;
@ foo bar
@ ;;
error recovered
@
```

Voici la liste des mots-clés connus par AUTO:

```
actions add-search-path and audit clear close comline
default-editor del-search-path display
edit end exclusion explore flags globals in include
let lisp load load-auto load-criterion nf-auto obs
parse parse-nf remove search-path
set show sort trace tta visible write
```

Il existe aussi des variables dont l'usage est réservé au contrôle du système. Ces variables et leur utilisation sont décrites plus loin.

3.2 La commande set, l'attribut show et la variable it.

Cette commande sert à associer une valeur à une variable.

```
@ set x = 3;
x : Integer
```

La commandes `set`, ainsi que toute commande du toplevel, rendent un résultat qui n'est pas affiché. Ce résultat est associé à la variable `it`. Pour obtenir l'affichage de cet objet, il faut précéder sa ligne de commande par le mot-clef `show`. Une autre possibilité est d'utiliser `it`, par exemple en demandant `show it;`.

```
@ set y = 4;
y : Integer
@ show it;
4 : Integer
```

On peut distinguer deux utilisations de la commande **set**:

- la modification des variables de contrôle du toplevel,
- l'association d'un objet à une variable.

Les fonctions traitant des termes et des automates sont décrites plus loin. Les variables de contrôle du toplevel peuvent prendre des valeurs booléenes (`true`, `false`) ou entières. Les noms de ces variables ainsi que leurs valeurs significatives sont détaillées pour chaque commande. Certaines de ces variables concernent le comportement général de l'interprète. Ce sont:

- la variable `timer`, un booléen contrôlant l'impression du temps de calculs de chaque commande ainsi que le temps d'impression du résultat, sa valeur par défaut est `false`.
- la variable `debug-gc` qui contrôle l'impression du message indiquant que le garbage collector de LELISP a récupéré une partie de la mémoire dont il dispose; ce message contient la quantité de mémoire récupérée, l'utilisateur peut ainsi se rendre compte d'un éventuel manque de mémoire. Cet indicateur vaut par défaut `false`.
- la variable `print-brackets`, qui contrôle le parenthésage des termes. Si elle vaut `false` (c'est la valeur par défaut), le système n'imprime que les parenthèses minimales (selon les règles de priorités, cf. section 2.2). Si elle vaut `true`, toutes les parenthèses sont imprimées.

Les autres variables de contrôle sont spécifiques à certaines commandes, et sont décrites en même temps que ces commandes. Les noms des variables de contrôle sont réservés pour cet usage, on ne peut leur affecter de valeur que du type prédéfini pour chacune.

Les commande décrites dans la suite de cette section sont les commandes traitant des fichiers de commande ou de trace, de la visualisation des objets calculés.

3.3 Les commandes *display* et *write*.

Ces deux commandes servent à imprimer un objet selon un format donné: `display` imprime sur l'écran et `write` sur un fichier.

`display` *ID [ID]*
`write` *chaîne ID [ID]*

La chaîne de caractères passée en argument à `write` est le nom du fichier où se fait l'impression demandée. Les deux autres arguments sont respectivement l'objet à imprimer et le nom du format d'impression désiré. Ce dernier argument est optionnel, tous les objets ayant un format d'affichage par défaut. Pour certains formats, la commande *write* ajoute au nom de fichier spécifié un suffixe (dépendant du format). Le format par défaut n'a pas de suffixe associé. Les formats des automates sont discutés dans la section 5.

Les impressions obtenues par *display* et par *write* sont strictement identiques, sauf que *write* ajoute un point-virgule en fin de fichier, pour pouvoir être relu par le système.

L'objet à imprimer peut être une variable toplevel ou certains éléments du système, désignés par des mots-clés. Les mots-clés connus sont:

- `flags` pour afficher tous les indicateurs du système,
- `globals` pour afficher toutes les variables liées au toplevel.

```
@ display flags;
debug-algo = False
debug-failure = False
```

```
debug-gc = True
debug-state = False
echo = False
print-brackets = False
timer = False
```

3.4 Les commandes *clear* et *remove*.

La commande `remove` prend en argument un identificateur et supprime sa liaison de l'environnement global.

```
@ set x=1;
x : Integer
@ x;
: Integer
@ remove x;
x Removed.
@ x;
Failure : get : undefined variable : x
```

La commande `clear` sert à reconstruire complétement tout le toplevel, c'est-à-dire qu'il remet tous les indicateurs à leur valeur par défaut et qu'il efface toutes les autres variables. L'écran est effacé, la bannière de AUTO est affichée, ainsi qu'un prompt `@`.

Dans la version actuelle, un bug oblige l'utilisateur à taper un retour-chariot excédentaire à la fin de la première commande qui suit un `clear`.

3.5 La commande *load*.

Cette commande sert à lire une suite de commandes toplevel dans un fichier. Sa syntaxe est:

`load` *chaîne*

Cette commande charge le fichier *chaîne*`.ec`, en le cherchant selon un search-path, on trouvera plus loin les outils de définition et de manipulations de celui-ci. Ces fichiers peuvent contenir des commentaires spéciaux qui permettent l'arrêt de la lecture des commandes et l'attente d'une action de l'utilisateur (taper return) pour reprendre cette lecture. Ces commentaires commencent par le caractère > en début de ligne et finissent par le le caractère < en début de ligne.

Les commandes `clear` et `lisp` (voir plus loin) sont interdites dans un "`load`".

3.6 Les commandes *audit* et *close*.

`audit` prend comme argument une chaîne de caractères. Cette commande ouvre le fichier dont le nom est cette chaîne suffixée par `.aud` qui va contenir toutes les commandes et les réponses faites par le système jusqu'à la prochaîne commande `close`.

```
@ audit "toto";
Auditing on toto.aud.

@ close;
File toto.aud closed.
```

3.7 Les commandes *lisp*, *comline* et *end*.

La commande **end** permet de sortir du système. La commande **lisp** permet de quitter le toplevel pour passer sous le toplevel de LELISP. Pour revenir sous le toplevel de AUTO, il suffit d'appeler la fonction **auto** en tapant **(auto)**. La commande **comline** a une chaîne de caractères comme argument; cette chaîne est passée comme commande à UNIX.

```
@ lisp;
? (de foo () (bar 'gee))
= foo
? (auto)
Comming back from lisp.

@ comline "pwd";
/users/meije/dvergami/philo
```

3.8 Les commandes *edit* et *default-editor*.

Ces commandes permettent d'appeler un editeur directement depuis le toplevel de AUTO. L'éditeur par défaut est **emacs**.

Syntaxe:

```
edit ''nom-de-fichier'';
default-editor ''nom-du-nouvel-editeur'';
```

Le nom-de-fichier passé à la commande **edit** peut être absolu, ou relatif au répertoire courant. Ce peut être aussi une chaîne vide.

La commande **default-editor** change l'éditeur courant.

3.9 Les commandes *search-path*, *add-search-path* et *del-search-path*.

La commande **search-path** ne prend pas d'argument et rend le search-path du système. Ce search-path sert à la recherche de fichier pour la commande **load**, et pour la fonction **include**. Il contient des chaînes de caractères qui sont des paths légaux pour le système (ici UNIX). Il est initialisé par défaut pour contenir le répertoire de travail courant (**"./"**) et certains répertoires d'exemples. Les deux autres commandes permettent de modifier ce search-path. Ces deux commandes ne prennent qu'un seul argument qui est une chaîne de caractères contenant le nom du répertoire à ajouter ou à supprimer. Cette chaîne doit se terminer par le caractère **"/"**.

Exemple:

```
@ search-path;
"./"
"/proj/ecrins/v1-6/test/"
"/proj/ecrins/v1-6/demos/"

@ del-search-path "/proj/ecrins/v1-6/test/";
"./"
"/proj/ecrins/v1-6/demos/"
```

4 Les algèbres de termes.

Ce chapitre décrit MEIJE0 une algèbre de termes proche de MEIJE, et NFAUTO, une algèbre pour les formes normales des termes de MEIJE0 qui nous intéressent. Les termes de MEIJE0 forment

la base de la construction des automates que AUTO permet de manipuler. MEIJE0 est construit à partir d'un groupe libre d'actions engendré par un ensemble de noms de signaux. Nous allons donc d'abord détailler la syntaxe de ces actions puis nous donnerons les opérateurs de MEIJE0.

4.1 Les actions de MEIJE0.

On construit un groupe libre à partir d'un ensemble de noms de signaux. Toute action est donc un "produit" de signaux de base et d'inverse de ces signaux. A partir d'un signal `x`, on obtient son inverse en écrivant `x?`, ses puissances positives et négatives en faisant suivre son nom respectivement par le nombre voulu de "`!`" ou "`?`". `x!` est équivalent à `x`. Il est de plus une action spéciale qui est l'élement neutre du groupe libre, celle-ci est notée `eps`.

Le produit de simultanéité est noté "`.`". Comme il est naturellement associatif, on lui associe une opération d'élévation à la puissance notée `^`. Ainsi `x^-3.y^2` est équivalent à `x???.y!!`.

Les morphismes sont représentés par la liste des signaux renommés et leurs images. Ce qui nous donne la syntaxe suivante pour les actions:

```
action                      : action-simple
                            | "(" action ")"
                            | action "." action
                            | action "<" renommage ">"
                            | action "^" ENTIER ;
action-simple               : "eps" | ID
                            | ID suite-de-! | ID suite-de-? ;
suite-de-!                  : "!" | "!" suite-de-! ;
suite-de-?                  : "?" | "?" suite-de-? ;
renommage                   : liste-de-renommage-simple ;
liste-de-renommage-simple   : renommage-simple
                            | renommage-simple ","
                              liste-de-renommage-simple ;
renommage-simple            : action "/" ID ;
```

L'opération d'élévation à la puissance a une priorité supérieure à l'opération de produit, l'opération de renommage étant l'opération associant le moins.

`manger!.lacher?^-2<manger.lacher? /lacher>`
est lue
`((manger!).(lacher?^-2))<manger.lacher? /lacher>`

⚠ L'analyseur lexical ne permet pas de coller n'importe quel caractère derrière une suite de ? ou de !. Les seuls caractères permis sont "`+ . * ^ :`" et les séparateurs bien entendu. C'est pourquoi on doit écrire `s!!.a?␣/b` et non `s!!.a?/b`.

On définit la forme normale d'une action comme étant le produit des projections non nulles sur l'ensemble des noms de signaux apparaissant dans l'action, triés par ordre alphabétique. Par exemple, l'action donnée plus haut a pour forme normale `lacher??.manger!!!`.

Nous utilisons dans la suite les notions de divisibilité d'une action par un signal et de compatibilité d'une action par rapport à une relation d'exclusion de signaux.

- un signal divise une action si il figure dans sa forme réduite; ainsi `a` divise `a?.b!` mais ne divise pas `a?.a!` qui est égale à `eps`,
- une relation d'exclusion de signaux est représentée par une liste de listes de signaux considérés comme incompatibles deux à deux. Une action vérifie une telle relation si aucun couple

de signaux la divisant n'est inclus dans une des ces listes. Soit R la relation d'exclusion représentée par `{{a1,a2,a3},{b1,b2,b3}}`, l'action `a1!.b2?` vérifie R et l'action `a1!.a3?` ne vérifie pas R. L'action `eps` vérifie n'importe quelle relation de ce type.

4.2 Les constructeurs de MEIJE0.

Les termes de MEIJE0 paramétré par le groupe d'actions décrit plus haut, sont construits à partir d'un ensemble de noms de variables au moyen des opérateurs suivants:

- les identificateurs,
- le processus inactif `0`,
- le séquencement d'une action et d'un processus `a:p`,
- le choix non-déterministe de deux processus `p+q`,
- le parallèle asynchrone de deux processus `p//q`,
- la restriction d'un processus par un signal `p\s`,
- le renommage d'un processus par un morphisme `p[R]`,
- les définitions récursives, par exemple `let rec x=a:x in x`.

La syntaxe concrète est la suivante:

```
processus             : ID | "0"
                      | action ":" processus
                      | processus "+" processus
                      | processus "//" processus
                      | processus "\" ID
                      | processus "[" renommage "]"
                      | "let rec" liste-de-definitions "in" processus
                      | "let rec" "{" liste-de-definitions "}" in" processus
                      | "(" processus ")" ;
definition            : ID "=" processus ;
liste-de-definitions : definition
                      | definition "and" liste-de-definitions ;
```

Les opérateurs associent dans l'ordre de priorité suivant:

```
// > \ > : > + > []
```

On a de plus la possibilité d'inclure dans une définition un terme décrit dans un fichier.

```
processus             : "include" chaine ;
```

La chaîne de caractères donnée en argument à `include` est le nom du fichier contenant le terme diminué du suffixe `".m0"`. L'écho du chargement est contrôlé par la variable `echo`. Si elle vaut `false`, le chargement du fichier est silencieux; si elle vaut `true`, les commandes contenues dans le fichier, et leurs résultats, sont imprimés au terminal (et éventuellement sur un fichier d'audit, s'il y en a un). Sa valeur par défaut est `false`.

La commande `parse` permet d'analyser un terme de MEIJE0 et de l'associer à un nom dans l'environnement du toplevel. On pourra écrire par exemple:

```
@ parse x= include "t1" // t2;
```

`t1.m0` est un fichier contenant un terme de MEIJE0 et `t2` est une variable qui peut être liée dans l'environnement global à un terme ou à un automate. Dans ce dernier cas, il y a une conversion implicite de l'automate en un terme.

A partir de maintenant, nous allons utiliser le classique problème des philosophes pour illustrer tout ce qui suit. Il s'agit de modéliser un repas de N philosophes disposant de N fourchettes pour manger de succulents spaghetti (des PANZANI peut-être). Mais nos philosophes ne sont capables de prendre leur pitance qu'avec deux fourchettes à la fois. L'attitude de chacun est donc: se saisir d'une fourchette, puis d'une autre, manger à sa faim, relacher les fourchettes et recommencer. On sent bien que si chacun prend une fourchette, plus personne ne pourra prendre une deuxième fourchette et tout notre beau monde va mourrir de faim. Il faut donc tenter de mettre un peu de discipline dans tout ça pour empêcher que tous les philosophes puissent avoir au même moment une fourchette chacun. Dans notre exemple, il y a trois philosophes et trois fourchettes.

Voici les différentes expressions utilisées pour décrire le réseau des philosophes et des fourchettes.

```
@ set echo=true;
echo : Bool

@ parse philo= include "philo";
let rec { dehors=entrer:dedans and
          dedans=prendre_g:prendre_d:mangeur
                 + prendre_d:prendre_g:mangeur and
          mangeur=manger:mangeur
                  + lacher_d:lacher_g.sortir:dehors
                  + lacher_g:lacher_d.sortir:dehors }
        in dehors;
philo : Process of meije0

@ parse fourchette=
let rec f0 = prendre?:f1 and f1 = lacher?:f0 in f0;
fourchette : Process of meije0

@ parse philo1=philo[prendre_3/prendre_g,prendre_1/prendre_d,
meije0>                  entrer_1/entrer,sortir_1/sortir,
meije0>                  lacher_3/lacher_g,lacher_1/lacher_d,
meije0>                  manger_1/manger];
philo1 : Process of meije0

@ parse f1=fourchette[prendre_1/prendre,lacher_1/lacher];
f1 : Process of meije0

@ % idem pour philo2, philo3, f2, f3
@ parse orgie=
meije0> ((philo1//f1//philo2)\prendre_1\lacher_1
meije0>   //(f3//f2//philo3))\lacher_3\prendre_3
meije0>                       \prendre_2\lacher_2;
orgie : Process of auto
```

4.3 Les termes en forme normale (l'algèbre NFAUTO).

La propriété pour un terme de MEIJE0 de dénoter un automate fini est indécidable. Il nous faut donc donner des contraintes sur l'écriture des termes pour pouvoir garantir le calcul des automates

associés. Certaines de ces contraintes sont nécessaires:

- les termes doivent être clos (c'est-à-dire ne pas contenir de variables libres),
- deux sous termes mis en parallèle doivent être clos,
- un sous terme renommé ou restreint doit être clos.

On ajoute à ces contraintes nécessaires deux autres contraintes:

- on ne doit écrire de sommes que gardées,
- un `let rec ... in ...` ne doit pas contenir de variables libres, les membres droits d'équation ne peuvent être que des sommes de séquencement d'actions et de variables ou bien `0`.

Les exemples donnés plus haut vérifient ces contraintes.
L'algèbre NFAUTO reprend deux opérateurs de MEIJE0:

- le renommage d'un processus par un morphisme `p[R]`,
- les définitions récursives,

Ainsi que deux nouveaux opérateurs:

- la somme gardée n-aire, dénotée `NSUM(a1:p1,...,an:pn)`, (la somme portant sur une liste vide de garde est notée 0),
- le parallèle restreint de deux processus, qui effectue en même temps la mise en parallèle et la restriction sur un ensemble de signaux `RPAR(p,q,[s1...sn])`.

4.4 La fonction *nf-auto* (:MEIJE0 → NFAUTO).

La fonction `nf-auto` appliquée à un terme de MEIJE0 permet de vérifier les contraintes ci-dessus; de plus, le résultat de cette fonction rend une forme normale où les définitions récursives portent uniquement sur des grammaires linéaires droites. Lorsqu'un terme ne respecte pas les contraintes précédentes, AUTO provoque un échec. Les différents messages d'échec sont:

```
FAILURE: forbidden construction in a let rec : parallel
FAILURE: non guarded sum
FAILURE: non closed term, free variable : x
```

Le premier type de message indique que l'on a utilisé un autre opérateur que la somme, la séquence ou zero dans une définition récursive. Le second type de message indique qu'une somme non gardée a été utilisée, le troisième qu'il existe une variable libre dans un sous-terme.

La mise en forme normale nécessite la création de noms de variables lorsqu'elle crée de nouvelles équations; ces noms sont simplement des numéros consécutifs, le premier créé étant 1. On aura par exemple:

```
@ show nf-auto philo1;
let rec
  dehors = SUM (entrer:dedans)
  and dedans = SUM (prendre_g:1, prendre_d:2)
  and 2 = SUM (prendre_g:mangeur)
  and 1 = SUM (prendre_d:mangeur)
  and mangeur = SUM (manger:mangeur, lacher_d:3, lacher_g:4)
  and 4 = SUM (lacher_d.sortir:dehors)
```

```
and 3 = SUM (lacher_g.sortir:dehors)
in dehors
[
prendre_3/prendre_g,prendre_1/prendre_d,entrer_1/entrer,sortir_1/sortir
 ,lacher_3/lacher_g,lacher_1/lacher_d,manger_1/manger]
: Process of nfauto
```

4.5 La commande *parse-nf*.

La commande parse-nf a une syntaxe identique à parse. Elle réunit l'analyse et la mise en forme normale d'un terme de MEIJE0.

```
@ set echo=true;
@ show parse-nf f = include "fourch";
let rec {f0=prendre?:f1 and
         f1=lacher?:f0} in f0;
let rec
  f0 = SUM (prendre?:f1)
  and f1 = SUM (lacher?:f0)
  in f0

f : nf-auto
```

4.6 La fonction *tta* (:NFAUTO → AUTOMATON).

L'automate que dénote un terme en forme normale est calculé par la fonction tta (abréviation de "term-to-automaton"). Cette fonction échoue si le terme n'est pas en forme normale. Dans le cas contraire, elle produit un automate en un temps qui peut être grand si on n'y prend garde. L'utilisateur dispose de deux indicateurs de contrôle lui permettant d'estimer la taille des automates construits ainsi que la vitesse de construction:

- debug-algo est un indicateur booléen. Lorsqu'il vaut true, un message est imprimé à chaque calcul de produit d'automates indiquant les tailles des deux membres du parallèle, un message donnant la taille du résultat est aussi produit à la fin du calcul.
- debug-state est lui aussi booléen. Lorsqu'il vaut true, lors du calcul d'un parallèle, un message est imprimé pour chaque création d'un nouvel état et pour chaque construction.

```
@ set debug-algo=true;
debug-algo : Bool

@ set Orgie=tta orgie;
par+ --> (7 10 8)(2 2 2)
par+ <-- (14 54 25)
par+ --> (14 54 25)(7 10 8)
par+ <-- (40 149 40)
par+ --> (2 2 2)(2 2 2)
par+ <-- (4 12 8)
par+ --> (4 12 8)(7 10 8)
par+ <-- (28 244 68)
```

```
par+ --> (40 149 40)(28 244 68)
par+ <-- (178 1203 50)
Orgie : Automaton
```

Les triplets de chiffres apparaissant dans la trace sont les tailles des automates construits exprimées en nombre d'états, nombre de transitions et nombre d'actions. On voit par exemple que l'automate correspondant au système de trois philosophes et de trois fourchettes a 178 états, 1203 transitions et 50 actions différentes.

4.7 La fonction *exclusion* (:NFAUTO # LIST OF LIST OF SIGNAL → AUTOMATON).

On est parfois amené à écrire des termes dont certaines actions potentielles contiennent des occurrences de signaux que l'on souhaite incompatibles. La fonction `exclusion` sert à construire l'automate que dénote un terme sous la contrainte d'une relation d'incompatibilité. Dans notre exemple, on peut construire l'automate global où on interdit à plusieurs philosophes d'entrer ou de sortir simultanément de la façon suivante:

```
@ set porte={{entrer_1,entrer_2,entrer_3},{sortir_1,sortir_2,sortir_3}};
porte : List of List of *

@ set ORGIE=exclusion(orgie,porte);
exclusion (7 10 8) avec ((entrer_1 entrer_2 entrer_3)
                         (sortir_1 sortir_2 sortir_3))
exclusion <-- (7 10 8)
exclusion (2 2 2) avec ((entrer_1 entrer_2 entrer_3)
                        (sortir_1 sortir_2 sortir_3))
exclusion <-- (2 2 2)
par&exclusion --> (7 10 8)(2 2 2) avec ((entrer_1 entrer_2 entrer_3)
                                        (sortir_1 sortir_2 sortir_3))
exclusion <-- (14 54 25)
exclusion (7 10 8) avec ((entrer_1 entrer_2 entrer_3)
                         (sortir_1 sortir_2 sortir_3))
exclusion <-- (7 10 8)
par&exclusion --> (14 54 25)(7 10 8) avec ((entrer_1 entrer_2 entrer_3)
                                           (sortir_1 sortir_2 sortir_3))
exclusion <-- (40 145 36)
exclusion (2 2 2) avec ((entrer_1 entrer_2 entrer_3)
                        (sortir_1 sortir_2 sortir_3))
exclusion <-- (2 2 2)
exclusion (2 2 2) avec ((entrer_1 entrer_2 entrer_3)
                        (sortir_1 sortir_2 sortir_3))
exclusion <-- (2 2 2)
par&exclusion --> (2 2 2)(2 2 2) avec ((entrer_1 entrer_2 entrer_3)
                                       (sortir_1 sortir_2 sortir_3))
exclusion <-- (4 12 8)
exclusion (7 10 8) avec ((entrer_1 entrer_2 entrer_3)
                         (sortir_1 sortir_2 sortir_3))
exclusion <-- (7 10 8)
par&exclusion --> (4 12 8)(7 10 8) avec ((entrer_1 entrer_2 entrer_3)
                                         (sortir_1 sortir_2 sortir_3))
```

```
exclusion <-- (28 244 68)
par&exclusion --> (40 145 36)(28 244 68) avec ((entrer_1 entrer_2 entrer_3)
                                                (sortir_1 sortir_2 sortir_3))
exclusion <-- (178 1115 34)
ORGIE : Automaton
```

Il n'y a plus que 34 actions différentes, ce qui a réduit le nombre de transitions à 1115.

5 Voir et analyser des termes et des automates.

5.1 Les formats d'impression des automates.

Les automates calculés dans AUTO peuvent être visualisés ou imprimés selon différents formats grace aux commandes `display` et `write`. Ceux-ci doivent permettre des visualisations diverses (standard ou résumée par exemple) ainsi que des décompilations pour "passer" les automates à d'autres formalismes (mec ...) et bien sûr un format spécifique à AUTO pour pouvoir réutiliser des automates déjà calculés. A ce dernier format est associée une commande permettant de recharger un automate.

5.1.1 Le format d'impression standard.

Ce format d'impression sert à visualiser tous les états avec leurs noms et la liste de leurs transitions. Les noms des états d'un automate global sont des listes des noms des états de chaque composant.

```
@ display PHILO;
7 states, 10 transitions, 8 actions.
State 0
(dehors)
-- entrer --> 1 (dedans)
State 1
(dedans)
-- prendre_g --> 2 (1)
-- prendre_d --> 3 (2)
State 2
(1)
-- prendre_d --> 4 (mangeur)
State 3
(2)
-- prendre_g --> 4 (mangeur)
State 4
(mangeur)
-- manger --> 4 (mangeur)
-- lacher_d --> 5 (3)
-- lacher_g --> 6 (4)
State 5
(3)
-- lacher_g.sortir --> 0 (dehors)
State 6
(4)
-- lacher_d.sortir --> 0 (dehors)
Initial-state 0
```

On trouvera à la section §5.2 un exemple où la structure du nom des états est utilisée pour comprendre dans quel état est le système global en fonction de l'état des ses composants.

5.1.2 Le format d'impression résumé (*short*).

Ce format d'impression indique simplement le nombre d'états, de transitions et d'actions différentes de l'automate.

```
@ display Orgie short;
size = 178 states, 1203 transitions, 50 actions.

@ display ORGIE short;
size = 178 states, 1115 transitions, 34 actions.
```

5.1.3 Le format d'impression MEIJE.

Ce format permet de décompiler un automate sous la forme d'un terme de MEIJE0. Ceci est fait à l'aide d'un `let rec`, dans lequel les états correspondent à des variables `s1,s2, ....` Lorsqu'il est utilisé avec la commande `write`, ce format correspond au suffixe `".m0"`.

```
@ display PHILO meije;
let rec
s0 = lacher_g.sortir:s1
and
s1 = entrer:s6
and
s2 = lacher_d.sortir:s1
and
s3 = lacher_g:s2 + lacher_d:s0 + manger:s3
and
s4 = prendre_d:s3
and
s5 = prendre_g:s3
and
s6 = prendre_d:s5 + prendre_g:s4
in s1
```

5.1.4 Le format d'impression AUTO.

Ce format n'est pas destiné à être lu par l'utilisateur, mais à conserver un automate pour une utilisation ultérieure, ou pour une utilisation par un autre système. Lorsqu'il est utilisé avec la commande `write`, ce format correspond au suffixe `".au"`. Il lui correspond une commande *load-auto*, qui permet de relire un automate dans ce format.

Si `x` est un objet de type `Automaton`, on l'écrit dans un fichier par:

```
@ write "PHILO" PHILO auto;
File PHILO.au written.
```

Puis voulant relire ce fichier plus tard (vraisemblablement dans une autre session):

```
@ load-auto x "PHILO";
x : Automaton
```

Le premier argument de cette commande est le nom de la variable à laquelle on veut associer l'objet lu. Noter que le suffixe ".au" est implicite pour la fonction *load-auto*.

5.2 La commande *explore*.

Lorsque la taille d'un automate est trop grande, l'impression de celui-ci devient soit impossible pour le sytème soit illisible pour l'utilisateur. La commande **explore** permet de ne visualiser qu'un état et ses transitions à la fois et d'explorer ainsi intéractivement les états qui nous intéressent. Cette fonction prend un automate en argument. Le premier état examiné est naturellement l'état initial de l'automate. Après avoir imprimé un état, **explore** demande un nouveau numéro d'état. On sort de cette boucle en tapant le caractère "." à une demande de numéro.

```
@ explore ORGIE;
State 0
(dehors f0 dehors f0 f0 dehors)
-- entrer_3 --> 1 (dehors f0 dehors f0 f0 dedans)
-- entrer_1 --> 2 (dedans f0 dehors f0 f0 dehors)
-- entrer_2 --> 3 (dehors f0 dedans f0 f0 dehors)
# 1
State 1
(dehors f0 dehors f0 f0 dedans)
-- eps --> 5 (dehors f0 dehors f1 f0 g234)
           4 (dehors f0 dehors f0 f1 g233)
-- entrer_1 --> 8 (dedans f0 dehors f1 f0 g234)
                7 (dedans f0 dehors f0 f1 g233)
                6 (dedans f0 dehors f0 f0 dedans)
-- entrer_2 --> 11 (dehors f0 dedans f1 f0 g234)
                10 (dehors f0 dedans f0 f1 g233)
                9 (dehors f0 dedans f0 f0 dedans)
# 20
State 20
(dehors f1 g229 f0 f0 dedans)
-- eps --> 71 (dehors f1 mangeur f1 f1 g234)
           67 (dehors f1 mangeur f0 f1 dedans)
           43 (dehors f1 g229 f1 f0 g234)
           42 (dehors f1 g229 f0 f1 g233)
-- entrer_1 --> 48 (dedans f1 g229 f1 f0 g234)
                47 (dedans f1 g229 f0 f1 g233)
                46 (dedans f1 g229 f0 f0 dedans)
                70 (dedans f1 mangeur f1 f1 g234)
                69 (dedans f1 mangeur f0 f1 dedans)
# 1434
state does not exist
# a;
** cy-explore : syntax error
error recovered
# .

@
```

Comme on l'a écrit en §5.1, les noms des états d'un automate global sont des listes des noms des états de chaque composant. Ainsi, `(dehors f0 dehors f0 f0 dehors)` correspond à un état du système `orgie` tel chaque philosophe est dans l'état `dehors` et chaque fourchette dans l'état =f0=.

5.3 La fonction *sort (: (*NFAUTO + AUTOMATON*)* → LIST OF SIGNALS*).*

Cette fonction s'applique sur tout terme en forme normale ou automate. Elle sert à savoir quels sont les signaux que peut "faire" un terme et permet en particulier de vérifier si une expression de réseau est bien écrite.

```
@ show sort ORGIE;
{entrer_1, entrer_2, entrer_3, manger_1, manger_2, manger_3, sortir_1,
 sortir_2, sortir_3}
  : List of Action
```

5.4 La fonction *actions (:* AUTOMATON →LIST OF ACTIONS*).*

On peut bien sûr appliquer la fonction `sort` à un automate. L'information que donne cette fonction est cependant bien trop sommaire alors que l'on a accès à toutes les actions différentes d'un automate. La fonction `actions` appliquée à un automate rend donc l'ensemble de ses actions. Appliquée à un terme qui n'est pas un automate, elle échoue en produisant un message du type:

```
Failure: actions must be applied on an automaton.
```

```
@ show actions ORGIE;
{entrer_1.manger_3.sortir_2, manger_3.sortir_2, entrer_3.manger_1.sortir_2
  , manger_1.sortir_2, entrer_2.manger_3.sortir_1, manger_3.sortir_1
  , entrer_3.manger_2.sortir_1, manger_2.sortir_1, entrer_1.manger_2.sortir_3
  , manger_2.sortir_3, entrer_2.manger_1.sortir_3, manger_1.sortir_3
  , entrer_1.sortir_2, sortir_2, entrer_3.sortir_2, entrer_2.sortir_1
  , entrer_3.sortir_1, sortir_1, entrer_2.sortir_3, entrer_1.sortir_3,
sortir_3
  , entrer_1.manger_2, manger_2, entrer_3.manger_2, entrer_2.manger_1,
manger_1
  , entrer_3.manger_1, entrer_2.manger_3, entrer_1.manger_3, manger_3, eps
  , entrer_2, entrer_1, entrer_3}
  : List of Action
```

5.5 La fonction *visible (:* NFAUTO # LIST OF SIGNALS →AUTOMATON*).*

La fonction `visible` prend deux arguments: un terme et un ensemble de signaux, elle calcule l'automate associé au terme réduit par rapport à un critère d'observation similaire à celui de Milner où tous les signaux qui ne sont pas dans l'ensemble donné en argument sont renommés en l'action invisible `eps`. Dans notre exemple, rendons invisible les actions d'entrer et de sortir, ce qui revient à rendre comme seule action visible celle de manger:

```
@ set repas={manger_1,manger_2,manger_3};
repas : List of *

@ visible(ORGIE,repas);
visible sur(178 1115 34) avec (manger_1 manger_2 manger_3)
```

```
observe+ --> (178 1115 4)
observe+ <-- (2 4 4)
visible <-- (2 4 4)
: Automaton

@ display it;
2 states, 4 transitions, 4 actions.
State 0
(1 f1 1 f1 f1 1)
     no transitions
State 1
(dehors f0 dehors f0 f0 dehors)
-- manger_2 --> 1 (dehors f0 dehors f0 f0 dehors)
-- manger_1 --> 1 (dehors f0 dehors f0 f0 dehors)
-- manger_3 --> 1 (dehors f0 dehors f0 f0 dehors)
-- eps --> 0 (1 f1 1 f1 f1 1)
Initial-state 1
```

On voit bien que le système réduit comporte un état de blocage. La structure de cet état de blocage nous montre qu'il se produit lorsque les trois philosophes ont tous pris leur première fourchette.

5.6 La fonction *trace (:* NFAUTO # LIST OF SIGNALS →AUTOMATON*)*.

La fonction `trace` a la même syntaxe que `visible`. Elle calcule l'automate déterministe reconnaissant le même langage que l'automate dénoté par le terme considéré. Ceci sous-entend que tous les états des automates produits par AUTO sont des états terminaux, que les signaux qui ne sont pas dans l'ensemble donné en argument sont renommés en `eps`, et enfin que l'action `eps` est vue comme le mot vide ϵ.

```
@ trace(ORGIE,repas);
: Automaton

@ display it;
1 state, 3 transitions, 4 actions.
State 0
()
-- manger_3 --> 0 ()
-- manger_1 --> 0 ()
-- manger_2 --> 0 ()
Initial-state 0
```

5.7 La fonction *obs (:* NFAUTO →AUTOMATON*)*.

La fonction `obs` (abbréviation de "observation") prend en argument un terme et rend l'automate associé réduit par rapport au critère d'observation de Milner. Cette réduction est faite sur tous les automates intermédaires. Si on ne veut faire la réduction que du système global, on doit d'abord le construire par `tta`.

Dans notre exemple, nous introduisons un protocole empêchant nos philosophes de pouvoir être tous les trois à table au même moment:

```
@ parse garcon=let rec libre=entrer?:occupe and
meije0>                        occupe=sortir?:libre in libre;
garcon : Process of meije0

@ parse service=garcon//garcon;
service : Process of meije0

@ parse banquet=(ORGIE[entrer/entrer_1,entrer/entrer_2,entrer/entrer_3,
meije0>                        sortir/sortir_1,sortir/sortir_2,sortir/sortir_3]
meije0>                  // service)\entrer\sortir;
banquet : Process of meije0


@ set BANQUET=obs tta banquet;
par+ --> (2 2 2)(2 2 2)
par+ <-- (4 12 5)
par+ --> (178 1115 16)(4 12 5)
par+ <-- (118 627 4)
observe+ --> (118 627 4)
observe+ <-- (1 3 4)
BANQUET : Automaton

@ display it;
1 state, 3 transitions, 4 actions.
State 0
(dehors f0 dehors f0 f0 dehors libre libre)
-- manger_3 --> 0 (dehors f0 dehors f0 f0 dehors libre libre)
-- manger_1 --> 0 (dehors f0 dehors f0 f0 dehors libre libre)
-- manger_2 --> 0 (dehors f0 dehors f0 f0 dehors libre libre)
Initial-state 0
```

Le système obtenu est conforme à l'idée que l'on a d'un banquet où tout le monde peut manger à sa faim.

5.8 Equivalence forte de deux automates: la fonction *eq (: (*AUTOMATON # AUTOMATON*)* → BOOLEAN*).*

La fonction *eq* teste l'équivalence forte de deux automates.

```
@ show eq(tta philo, PHILO);
True : Bool
```

5.9 Equivalence observationnelle de deux automates: la fonction *obseq (: (*AUTOMATON # AUTOMATON*)* → BOOLEAN*).*

La fonction *obseq* teste l'équivalence observationnelle de deux automates.

```
@ show obseq(obs philo, PHILO);
True : Bool
```

6 Les critères d'observation réguliers.

On va décrire rapidement l'utilisation dans AUTO des critères d'observation réguliers.

6.1 La syntaxe des critères.

Un critère d'observation "régulier" est une liste d'actions abstraites, chacune d'elles possèdant un nom et une expression régulière. Une expression régulière est formée en utilisant les opérateurs ":" (séquence), "+" (ou) et "*" (étoile) à partir d'expressions atomiques. Les expressions atomiques sont de la forme:

- a si a est une action, qui est la propriété "être égal à a",
- $/a$ si a est un signal, qui est la propriété "être divisible par a".

Une expression atomique peut aussi être la conjonction, la disjonction de deux expressions atomiques, ou la négation d'une expression atomique.
On a donc la syntaxe suivante pour les critères réguliers :

```
critere                  : liste_actions_abstraites
                         | "include" chaine ;

liste_actions_abstraites: action_abstraite
                         | action_abstraite "," liste_actions_abstraites ;

action_abstraite         : identificateur "=" expression_reguliere ;

expression_reguliere     : "(" expression_reguliere ")"
                         | expression_reguliere "+" expression_reguliere
                         | expression_reguliere "*"
                         | expression_reguliere ":" expression_reguliere
                         | expression_atomique ;

expression_atomique      :"(" expression_atomique ")"
                         | expression_atomique "and" expression_atomique
                         | expression_atomique "or" expression_atomique
                         | "not" expression_atomique
                         | action
                         | "/" signal ;
```

Les règles de priorité entre +, : et * sont:

```
* > : > +
```

6.2 La commande *parse-criterion*.

C'est la commande toplevel qui permet de lier une variable globale à un critère régulier. Elle s'utilise de la façon suivante:

```
@ parse-criterion crit=include "philo";
prendre_g=(entrer:prendre_g) + prendre_g,
prendre_d=(entrer:prendre_d) + prendre_d,
manger=manger,
```

```
lacher_d=(sortir.lacher_d) + lacher_d,
lacher_g=(sortir.lacher_g) + lacher_g;
crit : Criterion
```

Le suffixe `.crit` est implicite pour les fichiers chargés par `include`. Le critère précédent avait donc était édité dans le fichier `philo.crit`.

6.3 Observation d'un automate par un critère régulier : la fonction *quotient (: (*AUTOMATON # CRITERION*)* → AUTOMATON*).*

Si l'on a chargé dans l'environnement AUTO un automate (en général en chargeant un terme MEIJEO, puis en utilisant la fonction *tta*), on peut l'observer par un critère régulier. On utilise pour cela la fonction toplevel *quotient*, qui prend en arguments un automate et un critère.

```
@ quotient(PHILO,crit);
: Automaton

@ show it;
6 states, 9 transitions, 13 actions.
State 0
(dehors)
-- prendre_d --> 1 (2)
-- prendre_g --> 2 (1)
State 1
(2)
-- prendre_g --> 3 (mangeur)
State 2
(1)
-- prendre_d --> 3 (mangeur)
State 3
(mangeur)
-- lacher_g --> 4 (4)
-- lacher_d --> 5 (3)
-- manger --> 3 (mangeur)
State 4
(4)
-- lacher_d --> 0 (dehors)
State 5
(3)
-- lacher_g --> 0 (dehors)
Initial-state 0
: Automaton
```

7 Bibliographie.

[Au&Bo] D. Austry & G. Boudol, *"Algèbre de processus et synchronisation"*,Theoret.Comput. Sci. 30 p91-131 (1984)

[Bo] G. Boudol, *"Calcul de processus et vérification"*, Rapport INRIA 424 (1985)

[Mi] R. Milner, *"Calculi for synchrony and asynchrony"*, Theoret. Comput. Sci 25 p267-310 (1983)

[Pl] G. Plotkin, *"A structural approach to operationnal semantics"*, Report Daimi FN-19, Comput. Sci. Dept., Aarhus Univ. (1981)

[Ve] D. Vergamini, *"Verification by means of observational equivalence on automata"*, Rapport INRIA 501 (1986)

8 Résumé des commandes et fonctions de AUTO.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique